Alternatives for Constructing Computing Systems

Gio Wiederhold, Stanford University and ARPA
Stephen Cross, ARPA

Introduction

We address in this paper approaches to building large information systems, and contrast two paradigms. The paradigms can be associated with different business practices. Such business practices have an effect on the ability of independent entrepreneurs to participate in the software construction process. Practices also differ in countries such as Japan and the USA, although there is a good deal of overlap, and multinational operations can rarely be categorized at all.

With these two paradigms come alternatives in software construction practice, and different emphases on tools and standards. We will illustrate some of these in detail, and also describe research initiatives at ARPA which relate to these paradigms. We label these alternatives as the Software Engineering Model and the Service Model, and characterize the distinctions.

The intent of the paper is not to establish one approach as being superior to another, but rather to develop an understanding of the differences. Such an understanding should also make it simpler for these approaches to interoperate. Practical systems are rarely purely of one persuasion now, and are less likely to be so in the future, as large, practical systems will span boundaries of countries and of software practice. The Service Model opens up opportunities for individual entrepreneurs, including those that best work at home, perhaps because of limited mobility. However, integration and efficiency of software is likely to be greater when following the Software Engineering Model.

1. Building and maintaining large systems.

The construction of large software systems remains a major challenge. Large software systems require interaction of many specialists in their definition, the transmission of specifications to implementors, and the commitment or binding of the programs to specific data structures, processing sequences, and data storage schemes. When the software is finally completed it may not satisfy the user's formal specifications, and even less the user's informal expectations.

Even though impressive advances are being made in formal verification, so that today the revification of an entire operating system kernel can be attempted [Bevier:93], such a task is still a major challenge, specifically the generation of adequate specifications.

Larger systems, even if composed of separately created and tested components, still require much testing, and the impossibility of comprehensive or even thorough testing of large systems is well accepted [Parnas:85]. Furthermore, we do agree with Dijkstra that testing can only prove the presence of errors and not their absence. But that insight does not provide guidance, as the scale of the examples analyzed in [Dijkstra:76] shows.

In response to such problem statements the U.S. Department of Defense, which was concerned about building large systems, initiated a program to develop support for Software Technology for Adaptable and Reliable Systems [STARS:83]. This project eventually focused on building tools for large software projects, often in conjunction with the Ada [Royce:90] language. Ada is carefully structured to catch as many problems as possible syntactically. It supports consistency among modules by sharing descriptions and packages. Experience has shown that, indeed, errors in software can be reduced by these means. However flexibility tends to be reduced, because of the careful poreplanning needed to design the shared data structures, and the subsequent binding of programs to such structures.

The best language can not guarantee that large software will be written well. The architecture and the process of software creation, and systems construction and adaptation is much more critical.

So how do we build the large, operational systems we have today? The present practical paradigm is to adopt traditional engineering practice, i.e., do 'software engineering'.

The Software Engineering Paradigm

Modern software engineering is based on the spiral model [Boehm:88]. The essential steps are

(1) an initial statement of requirements
(2) a specification, preferably (but rarely) in a formal language
(3) the creation of a prototype
(4) an assessment of the prototype, preferably with the user that set the requirements
(5) a refinement of the requirements
(6) preparation for a new round which reiterates steps 1-5.

At some point the system achieves a quality of operation that at step (3) we have a functional prototype, that can actually be used to replace earlier processes. The process can be conceptualized as one of increasing binding. During each iteration the binding is increased by committing informal requirements to real, implemented code objects [Wiederhold:77]. At each iteration the scope or breadth of the prototyped system increases.

During each round of the spiral (step 4) tests must be performed. Two types of evaluation are needed:

1. A final exam, to make sure that the objectives for this round are satisfied, and that the users will accept the results of this round
2. A regression test, where all the tests performed in previous rounds are replayed, to assure that other capabilities have not been lost. Note that the assumption is here of continius growth of capability.

If, in the development, older capabilities are to be omitted, then the regression tests must be adapted, or else they will fail.

Minor changes can sabotage regression tests. Wording on an interface to improve communication with human users can lead to errors, and if not dealt with propoerly, can cause whole sections of the regression test to be skipped. In general, the interfaces that we now prefer for rapid human interactions, with graphics, mouse-pick selection, and perhaps voice commands for input, blinking icons and speech for output are very difficult to integrate into regression tests.

A possible solution to the complexity of human interfaces is to separate the user-friendly aspects of large programs from the functional code, and provide a formal, 'machine-friendly' interface that permits effective testing. The actual user-fiendly interface can then be placed on the user's workstation. Such an interface will also provide flexibility for individual users. Such flexibility can enhance system adaptation to the special needs of users who are partially disabled, since now only local code has to be changed to adapt systems to specific requirements and capabilities.

With the spiral model of software engineering all modules of a system must be developed to some level to get to an adequate functional prototype. However, full synchronicity in development of a large software system is not needed. The objective of the spiral approach is risk reduction, meaning that the module which embodies the greatest risk should receive the most attention.

If the risk is performance, then the most intensely used modules must be made reasonable complete, but also supplied with the data volumes and interaction rates that are to be expected from the requirements specification.

If user acceptance is considered the greatest risk factor, then the (probably graphic) user interface will need the most initial attention. However, only presenting a pretty scenario will not lead to a functional prototype, and will lead users of only iterations frustrated because their needs are not being met. The availability of functional prototypes can greatly improve the consensus setting process, since it binds the abstract concepts to real objects. If these prototypes can be integrated into the daily work of the users then many problems and opportunities for improvement will be discovered early, and can be implemented at a reasonable cost in a succcessive round of the spiral.

The close cooperation engendered by such a modern software engineering approach leads to software that will be highly effective in its domain. A domain focus is needed to avoid distracting issues. Also the scope of the system should not be broader than the domain expertise of the participants.

Since large systems required continued maintenance it is important that the group of participants, or at least a skeleton version of it remain available.

Under maintenance we do not focus on bug-fixing, i.e., failure of the system function according to the known requirements, but rather the continued adaptation to changes in actual user needs, physical environments, government and business rules, and the natural growth of coverage in successful applications.

2. The Servive Paradigm

We characterize the alternative paradigm as a 'service' approach. In the service approach the emphasis is on providing services at several layers. The services may be provided by relatively autonomous vendors, who wish to maximize their profit, be it financial or professional recognition.

Examples of software operations on a service model are databases. We can use this example to identify important components need for a service paradigm.

(1) The service operates autonomously, often remotely, through a formal interface

(2) There is a clear and simple model of their functionality for the user, which hides performance-oriented complexities, such as indexing, compression, backups, and consistency management.

A problem with the service model is that vertical integration is relatively ad hoc. Each layer will induce problems of impedance mismatch. The value added in each layer has to be sufficient to overcome these liabilities.

A vendor of a service has to strike a balance in order to maximize market share. A service can be broad, i.e., to try to serve many customers. Mathematical routines are most commonly cited [ARPA software strategy] statistical analysis programs are an example at a larger scale of relatively broad services. Some sophisticated statistical packages include tests for the validity fo the analysis, and hence require the user to provide a model of the problem together with the data sets. A number of successful businesses provide statistical programs, some with consulting services specific to their domain of applicability. Soon domain specialization reduces the breadth of applicability, but increases the value added by the service. For instance, programs to provide survival analyses or analyses of medical procedures include simulation of normal populations of similar age distributions for comparison.

An example of further specialization is the NASA-sponsored library for analysis of flight parameters among object in space [Acton:93]. While the underlying basis is a fairly straightforward geometrical analysis of flightpaths, the routines take the speed of light into account when objects in space are being viewed, because the great distances involved. This library is served by a formal composition tool, which understands some of the constraints that must be obtained when assembling systems using these modules [Lowry:93].

Simulations

A great service potential exists for simulations. Simulations require specialized expertise in their design, explicit domain models, and extensive computational resources, but their interfaces can be relatively simple.

Financial simulation project sales into the future, given price information and market data. Manufacturing simulations quantify insight into product cost, and can be extended to include maintenance costs. Business simulation assess the future, given alternative investment decisions. Medical simulations can provide statistical or functional prognoses.

Service Interaction

The future of the service model will depend critically on future network infrastructure. Today the provision of added-value services is only feasible for fairly large scale systems. To obtain a service, a potential user has to establish an account with a service, obtain a credit rating or deposit a prepayment, and pay bills on a monthly basis, all via traditional paper and mail mechanisms. An exception is the Electronic Distribution Service service by EI Technology, which is ready to accept charges to credit cards on-line, without pre-arrangements. Digital credit card signatures are accepted and safeguarded by a mutual public key encryption scheme [WooLam:92]. Of course, today the credit card company still mails out a bill and expects a paper check in return, but that check covers all transactions with credit company for the month, greatly reducing the billing cost per transaction.

Easy and low cost incremental billing will be an essential aspect of a service model using computer networks, permitting many vendors to enter and compete in the market. Another crucial aspect is following standards, not necessarily ones that are now established, but those that are emerging from the conventions that are now coming into use [Neches:91].

Execution sites

As adequate protection methods for software operating on networks are developed, the service model employing vendor can be extended to execution on sites other than the vendor's. Since communication will always be an overhead versus execution on the user's home medium, it should be possible to support concepts as leasing of software for use, possibly with a per-use change. Sometimes the optimal execution site will be neither the machine of the vendor, nor the machine of the user of the software, but rather a machine near the data resources, so that communication bandwidth is reduced for data analysis programs. Only the results need to be shipped to the user and the billing information goes to the software vendor. A program requiring intense computation is best moved to be executed on a high-performance resource within the network. That site can either bill the user or the vendor for its cycle use.

Meta Information

In the service model, meta information will become crucial. It must be possible for users to advertise their services, and specify them sufficiently formally so that use and composition decisions can be supported by second level tools and services.

Optimization of systems assembled in a service model also requires cost predictions, so that service modules must also be able to provide cost information, in terms of dollars as well as response time and output volumes [Wiederhold:92]. While prices can be set by the vendor and be very clear to the user, the performance of computer programs can be hard to predict, and require a considerable amount of complementary code in the service program. For some complex services simulations may be needed, for others it should be possible to employ table lookups. Such tables are now available in printed form for some classes of programs, as commercial sort routines.

Comparing the Paradigms

When a computation has always been seen in one paradigm, it is hard to visualize it any other way. This problem is commonly seen today in the move to object-oriented programming. The service paradigm further scales the object-oriented approach to give the modules yet more autonomy, whereas in a single object-oriented program a compiler will have access to the internal of the objects being used.

Moving from a service to a software engineering approach requires gaining a deeper understanding of the internals of the services, so that the expected tight coupling can be realized.

If we can have computers and formal descriptions that permit full integration of service function into programs, then the paradigm difference may mean little to the operational user. The differences are considerable for the vendor. A servive vendor is quite independent, and the model can empower indivivual enterpreneurs. For people that prefer to work at home, perhaps because they have limited mobility themselves, or responsibilities to care for others, the provision of services over networks can provide new opportunities. It empowers them with the capability for tele-commuting, and aspect of seen in future socieiies, but not yet specifically considered from the viewpoint of software architecture.

It will take some time to establish the environment for service computing, although all the prerequisites exist at this time. In the meantime, most large systems will be constructed in a mixed mode, using some services explicitly, where they are available, and assembling the remainder by iterative design.

Conclusion

Construction of large software systems can follow approaches that differ in a deeper sense than choice of computer, system environment, tools, or languages. In this paper we distinguish two paradigms, labeled software engineering and service provision, respectively. We describe these alternatives, without indicating one as better than the other, since the crucial decisions will be based on system objective and available resources.

However, the understanding gained in characterizing those distinctions can help in making decisions in system construction in a more explicit way

References

[Acton:93] C.H. Acton: "Using the SPICE System to Help Plan and Interpret Space Science Observations"; *Proceedings of the Second International Symposium on Ground Data Systems for Space Missions Operations*, Pasadena, California, November 16-20 (JPL Publication No. 93-5, March 1, 1993).

[ARPA:91] *Department of Defense, Software Technology Strategy*; Dec.1991, prepared for the Director of Defense Research and Engineering.

[Bevier:93] William R. Bevier and Paul Dale: *Testing the Mach Kernel for Compliance with a Mathematical Specification*; Computational Logic, Inc., Feb.1993.

[Boehm:88] Barry W. Boehm: "A Spiral Model of Software Development and Enhancement"; *IEEE Computer*, Vol.21 No.5, May 1988, pp.61-72.

[Parnas:85] David Parnas: "Software Aspects of Strategic Defense Systems"; *Comm.ACM*, vol.28 No.12, Dec.1985, pp.1326-1335.

[Dijkstra:76] E.W. Dijkstra: *A Discipline of Programming*; Prentice-Hall, 1976.

[Lowry:93] Michael Lowry, Tom Pressburger, Mark Stickel, Ian Underwood, and Richard Waldinger: *Deductive Composition of Astronomical Software*; Technical Report, SRI International and NASA Ames Research Center, 1993.

[Neches:91] Robert Neches, Richard Fikes, Tim Finin, Tom Gruber, Ramesh Patil, Ted Senator, and William R. Swartout: "Enabling Technologies for Knowledge Sharing'; *AI Magazine*, Vol.12 no.3, Fall 1991.

[Royce:90] W.E. Royce: "TRW's Ada Process Model for Incremental Development of Large Software Systems", *Proc. 12th Internat. Conf. on Software Engineering*, IEEE, Nice, Mar.1990.

[STARS:83] Department of Defense: *Software Technology for Adaptable, Reliable Systems (STARS)*; April 1983.

[Wiederhold:77] Wiederhold, Gio: *Binding in Information Processing*; Stanford University, Computer Science Dept. Report, STAN-CS-81-851, May 1981.

[Wiederhold:92] Wiederhold, Gio: "Model-free Optimization"; *DARPA Software Technology Conference 1992*, April 1992, Meridien Corp., Arlington VA, pp.83-96.

[WooLam:92] Thomas Y.C. Woo and Simon S. Lam: "Authentication for Distributed Systems"; IEEE Computer, Vol.25 no.1, Jan.1992, pp.39-52.